Serendipity: A Distributed Computing Platform for Disruption Tolerant Networks

January 2011
GT-CS-11-08

Cong Shi*, Vasileios Lakafosis†, Mostafa H. Ammar*, Ellen W. Zegura*
*School of Computer Science
Georgia Institute of Technology
{cshi7, ammar, ewz}@cc.gatech.edu
†School of Electrical and Computer Engineering
Georgia Institute of Technology
vasileios@gatech.edu

ABSTRACT

The opportunistic or disruption tolerant networking (DTN) paradigm shows up in a variety of settings, from military to disasters to the developing world to deep space; anywhere that fixed infrastructure is either unavailable or expensive. Simple messaging applications have substantial value for communication and coordination. We posit that these settings can also leverage applications that are computationally complex and will benefit from distributed computing to take full advantage of nearby computational resources. Computing over these networks is not trivial, however, since network disconnections are common and persist over many time scales.

In this paper we present the design and implementation of Serendipity, a general purpose distributed computing platform designed to run over a DTN. We have designed a simple but powerful job structure that is suitable for such an underlying network. As opposed to traditional distributed computing platforms in data centers and clusters, where a central master is used to allocate tasks and monitor the working nodes, Serendipity relies on the collaboration among DTN nodes on these functionalities. Smart task allocation algorithms are designed to disseminate tasks among mobile devices by accounting for the special properties of DTNs. The extensive evaluation of our system on Emulab demonstrates that Serendipity efficiently speeds up various kinds of distributed computing jobs by a factor of 2.3 to 10.1 in a diverse set of DTN environments.

1. INTRODUCTION

The opportunistic or disruption tolerant networking (DTN) paradigm shows up in a variety of settings, from military [26, 30, 27] to disasters [15, 17] to the developing world [28, 12, 6, 16, 33] to deep space [9, 20]. These settings share the characteristic that fixed infrastructure is unavailable, highly unreliable, or expensive. Further, the communication links are subject to long delays or disruptions that mean network partitions are common. The DTN research community has worked for nearly a decade on algorithms, protocols and architectures to accomplish communication in these challenged environments, with much of the work having focused on the problems of architecture, routing and data transfer [14, 36, 21].

DTN application development has lagged routing and data transfer protocol development. The reasons for this are several fold, but likely include (1) a natural progression for networking researchers to focus first on infrastructure development and then on uses for infrastructure and (2) a lack of operational DTNs outside of the military context to provide driver applications and testing opportunities. Progress has been made on applications for networks that are hybrids with part of the network experiencing disruptive connectivity (usually confined to the last hop) and the rest consisting of reliable infrastructure, such as in a vehicular setting where wifi access connectivity is intermittent [5]. In these settings, however, the focus is often on adapting standard applications such as Internet web access and search to compensate for the last hop [6].

We posit that an additional contributing factor to the lack of application development may be the challenge of envisioning applications that are able to tolerate the disruptions present in a DTN. To make progress on application support in pure DTNs, we identify the following scenarios:

Situational Awareness: A soldier in the field must make a decision about whether to fire or not. The situation contains danger and uncertainty. Real-time images collected by this soldier and by others nearby contain information relevant to the decision such as location of target, presence of civilians, availability and location of backup troops, etc. While useful, the image data is incomplete and noisy. There would be great value in the ability to run, in real-time, image processing algorithms to aid situational awareness and decision making.

Real-time Planning and Coordination: Bystanders are on the scene of a natural disaster. They possess varied skills and resources relevant to the situation, such as first aid training, strength, water, cell phones. They must quickly make decisions about who will do what, in what order, and with what resources. This is a logistics planning and coordination problem where the ability to run optimization code to guide decisions might make a real difference to survivors.

Natural Language Support: A relief worker in the developing world travels to villages, each with their own local language dialect. She has some fluency in the national language. A machine learning-based, natural language processing program would allow her to better communicate and help ensure accuracy in the dialog. Examples could derive from references to the national language. The prevalence of cell phones even in the developing world suggests some availability of computing resources.

The reader may be surprised to see these examples suggested for suitability in a DTN, however we assert that they represent a useful and interesting class. The key characteristics of these applications vis $\grave{a}$ vis a DTN are: (1) they are best effort in the sense that they are intended to improve high-level outcomes but are not absolutely necessary; (2) they have some computationally complexity, but if coded properly, can take advantage of whatever computation is available (more computation can produce more accurate results while less computation is still useful); (3) they have timing requirements that call for expedient processing. The network environment for each of these scenarios meets the pure DTN characteristics, namely no infrastructure, dynamic and unstable wireless links, and mobility that causes nodes to move out of communication range of other nodes for arbitrarily long periods of time.

Taken together, these features suggest that distributed computation that is cognizant of the underlying network environment can be a powerful application enabler in DTNs. In this paper we present the design and implementation of Serendipity, a distributed computing platform designed to run over a DTN. We explicitly focus our attention on pure DTNs (hereafter called DTNs) where there is no infrastructure present. These environments are significantly challenging, and understanding solutions in this context provides a basis for considering hybrid DTN-infrastructure environments in the future.

Serendipity includes four novel ideas for accommodating an underlying dynamic and unpredictable network. First, the building block for computation is amenable to "maximally local" distribution of tasks, concentrating computation on nodes that are well reached from an initiating node. This approach allows the initiator to carefully track completion and connectivity and easily recover from network changes during computation. Second, fully general directed acyclic graph (DAG) job structures are supported but each component of the DAG is scheduled on-demand to make best use of the current network conditions at the time the computation is needed. On-demand scheduling helps reduce the penalty associated with launching computation on nodes that subsequently move out of connectivity. Third, tasks within a job are prioritized so that when a single node executes multiple tasks, it does so to miminize overall completion time. Finally, our task assignment algorithms make use of any available information about the future, including estimates of task completion time and information about future network contacts, in settings where that is available.

The paper is organized as follows: we start with the discussion of the challenges and a suitable job model of distributed computing in DTNs in Section 2; a system overview of Serendipity is provided in Section 3; the task allocation and scheduling algorithms are described in Section 4 and 5, respectively; the implementation details are discussed in 6; the system is evaluated in Section 7; the related work is presented in Section 8; we conclude this paper and discuss our future work in Section 9.

2. DISTRIBUTED COMPUTING IN DTNS

In this section we discuss how to support distributed computing in a DTN environment. The discussion begins with a description of the network model. Then we analyze the challenges of distributed computing in DTNs and briefly outline the aspects of Serendipity that address these challenges. At the end of this section we describe the basic computational block that we use for job representation; this block provides key features that accommodate the DTN setting.

2.1 Network Model

A DTN is composed of a set of nodes with computation and communication capabilities. The network connectivity is intermittent leading to a frequently partitioned network. Every node can execute computing tasks, the number of which is constrained by its resources, such as central processing unit (CPU) capability, memory and hard disk size, and available energy[1]. The period of time during which two nodes are within communication range of each other is called a *contact*. During a contact nodes can transfer bundles [32], i.e. DTN packets, to each other. Both the duration and the transfer bandwidth of a contact are limited. A node can deliver bundles to a destination node directly if the latter is within radio range or, otherwise delivery occurs through routing via intermediate nodes. There are some variants of the general DTNs. In some special DTNs, such as those formed by smart phones, a low-capacity control channel (e.g., over a 3G cellular network) is available for metadata sharing. Otherwise, nodes can only exchange control information during the intermittent contacts. In addition, in some DTNs, such as space satellite networks, the node mobility patterns are predictable and, thus, their future contacts are also predictable. All these variants are taken into consideration in our design.

2.2 Challenges for Distributed Computing

Distributed computing involves the execution of computationally complex jobs through the cooperation among a set of devices that are connected by a network. Distributed computing performance relies on how well the jobs are matched

[1]We do not explicitly include power awareness in this paper, however we believe our framework can accommodate power considerations with modest changes to the task assignment algorithms. At an extreme, a node with limited power can be considered unreachable.

to the underlying environment, including device and network capabilities.

A major class of distributed computing jobs, supported by mainstream distributed computing platforms such as Condor [34] and Dryad [19], can be represented as DAG. As in the example shown in Figure 1, the vertices are programs and the directed links represent data flows between two programs. A traditional distributed computing platform maps the vertices to the devices and the links to the network so that all independent programs are executed in parallel and they transfer the output quickly to their children. In addition, a central master uses the network to coordinate and monitor the program execution on devices in a timely manner.

The intermittent connectivity in DTNs poses three key challenges for distributed computing. First, because the underlying connectivity is often unknown and variable, it is difficult to map computations onto nodes with an assurance that the necessary code and data can be delivered and the results received in timely fashion. This suggests a conservative approach to distributing computation so as to provide protection against future network disruptions. Given that the network bandwidth is intermittent, the network is more likely to be a bottleneck for the completion of the distributed computation. This suggests scheduling sequential computations on the same node so that the data available to start the next computation need not traverse the network. Third, when there is no control channel, the network cannot be relied upon to provide reachability to all nodes as needed for coordination and control. This suggests maintaining local control and developing mechanisms for loose coordination.

In response to these challenges, Serendipity has the following features. First, we simplify the job structure and emphasize parallelism opportunities. Our basic job component is described in the next subsection. Second, we simplify and distribute the control of the job execution, as described in Section 3. Third, we design task allocation strategies that take advantage of all available information about future network contacts and estimated task completion time, as described in Section 4 and 5.

2.3 A Job Model for DTNs

Our basic job component is called a *PNP-block*. As shown in Figure 1, a PNP-block is composed of a *pre-process* program, n parallel *task* programs and a *post-process* program. The pre-process program processes the input data (e.g., splitting the input into multiple segments) and passes them to the tasks. The workload of every task should be similar to each other to simplify the task allocation. The post-process program processes the output of all tasks; this includes collecting all the output and writing them into a single file.

The PNP-block design simplifies the data flow among tasks and, thus reduces the impact of uncertainty on the job execution. All pre-process and post-process programs can be executed on one initiator device, while parallel tasks are executed independently on other devices. The communication

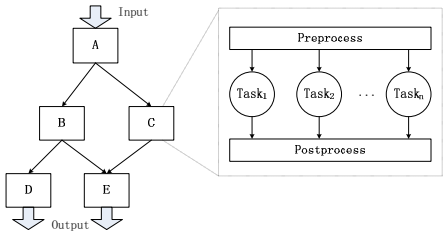
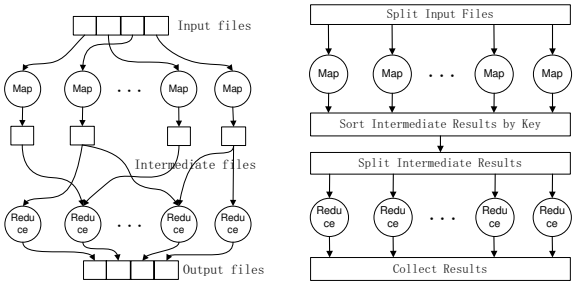


Figure 1: A Serendipity job is a directed acyclic graph (DAG), the vertices of which are PNP-blocks. Every PNP-block consists of a pre-process, a post-process and n parallel tasks. All the tasks are the same program with different input data.



(a) MapReduce model (b) Implementing MapReduce in Serendipity

Figure 2: MapReduce can be implemented in Serendipity with two sequentially connected PNP-blocks.

graph becomes a simple star graph. The data transfer delay can be minimized as the initiator device can simply choose nearby devices to execute tasks. In contrast, it is much more difficult for a complicated communication graph, such as the complete bipartite graph used in MapReduce [11] and Dryad [19], to achieve low delay in DTNs because the optimization problem associated with mapping the general graph onto a DTN is complex.

The single PNP-block job is an important class of distributed computing jobs often called embarrassingly parallel and useful in many applications, among which are BOINC [2], SETI@home [3] and parameter sweep [10]. All Serendipity jobs are graphically represented by a DAG of PNP-blocks, providing as much computational expressiveness as a regular DAG. For instance, as shown in Figure 2, the MapReduce model [11] can be implemented with two sequentially connected PNP-blocks, corresponding to the map phase and the reduce phase, respectively.

3. SYSTEM OVERVIEW

Figure 3 shows the high-level architecture of Serendipity. Every node in Serendipity has a *job engine* process, a *master* process and several *worker* processes. The number of worker processes can be configured, for example as the number of cores or processors of the node. Serendipity also needs access to the contact information database shared with other DTN mechanisms for better task allocation.

To submit a job to Serendipity, a user needs to provide a script specifying the job DAG, the programs for all PNP-blocks and the input data to the job engine. The script is submitted to the *job parser* for basic checking and processing.

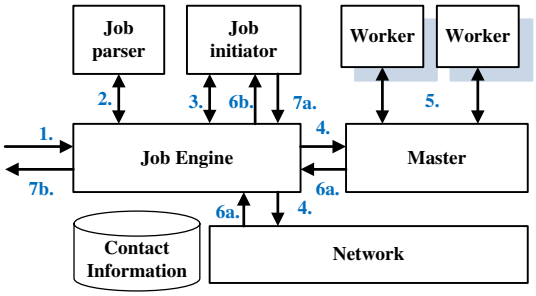


Figure 3: High-level Architecture of Serendipity. After receiving a job (1), the job engine parses the job (2) and starts a job initiator, who will initiate a number of PNP-blocks and allocate their tasks (3). The job engine disseminates the tasks to either local or remote masters (4). After a worker finishes a task (5), the master sends back the results to the job initiator (6a, 6b), who may trigger new job PNP-blocks (3). After all results are collected, the job initiator sends back the final results to the user (7a, 7b) and stops.

If everything is correct, the job engine will launch a new *job initiator* responsible for the new job. The job initiator stores the job information in the local storage until the job completes. All PNP-blocks whose parents have completed will be launched by running their pre-process programs on a local worker and assigning a worker to every task. The worker can be a single node, a set of candidate nodes, or a wildcard. Then these tasks will be sent to the job engine who is responsible for task dissemination. The $master$ is responsible for monitoring the task execution on workers. After receiving a task from the job engine, it starts a worker for it. When the task finishes, the output will be sent back to the job initiator. The job initiator returns the final output to the user when the entire job completes.

In the following subsections, we will discuss the design of job initiator, job engine, job parser and master in Figure 3.

3.1 Job Initiator

Every job initiator controls and monitors the job execution. It records the job in local storage, schedules and allocates tasks and collects the results.

When a job initiator starts, it first assigns a priority to every PNP-block and its tasks according to a certain strategy. When two tasks of a job are allocated to the same node, the task with the higher priority will be executed first. Explicitly setting priorities to tasks in Serendipity helps disseminate and execute tasks as soon as possible while keeping their relative urgency acknowledged by other nodes. More details of priority assigning will be discussed in Section 5.

Afterwards, the job initiator starts all the PNP-blocks, whose parents have completed, and allocates their tasks to proper workers. As opposed to traditional distributed computing platforms, which allocate all tasks at once, task allocation in Serendipity is conducted gradually as needed, i.e., once for a PNP-block. The execution of every PNP-block is simplified to a process of disseminating, executing and collecting the output of its tasks. The task allocation algorithms only need optimize this process without considering the complicated communication cost among tasks or mapping the job DAG to the intermittently connected mobile devices to minimize the overall communication time and execution time. Therefore, it will be significantly simplified and robust in the face of exceptions and prediction errors. In addition, job initiator does not allocate the tasks until they are to be disseminated out, reducing the impact of uncertainties in DTNs. Moreover, no communication among tasks is necessary in our design, avoiding DTN bottlenecks. The detailed task allocation algorithms will be presented in Section 4.

Meanwhile, the dependence among PNP-blocks considered in the task allocation of traditional distributed computing platforms is not considered in the our task allocation algorithms. Instead, this dependence is enforced by assigning different priorities to the PNP-blocks, as discussed above.

The job initiator also periodically records checkpoints for failure recovery. More details are provided in Section 6.

3.2 Job Engine

The job engine is primarily responsible for disseminating tasks, scheduling the task execution for the master and transferring the task output to the initiators through the collaboration with other job engines.

Every job engine has a dissemination list for tasks to be transferred to other nodes, a task list for tasks to be executed locally and an output list for unacknowledged task outputs. The task dissemination process relies on the underlying DTN routing protocols to intermittently transfer the task to its destination when connectivity permits. The task list has a two-dimensional structure. Ever task is put into a priority queue corresponding to its job, which sorts tasks according to their priority. Meanwhile, an entry in the task list is added and points to the queue. The node owner can set arbitrary priority policies for jobs from various users. A common policy is first in first out (FIFO). It is worth noting that tasks arriving late may be executed early by replacing early tasks with low priorities from the same job. The output collection process also relies on the underlying routing protocols to transfer data.

There are tasks whose destinations are a set of nodes or wildcards. These tasks will be added to both the disseminating list and the task list if current node is one of the destinations. Whenever this kind of tasks are disseminated out or executed, they will be removed from both lists. For those tasks with a wildcard, the job engine will also participate into the task allocation process as these tasks rely on the collaboration among job engines to find proper workers to execute them. More details are discussed in Section 4.

3.3 Job Parser

The job parser is responsible for parsing the job scripts and constructing the job DAGs. The basic script language is similar to that of Condor [34]. Every PNP-block should be declared by specifying the pre-process program, the task program, the number of tasks, the post-process program, its

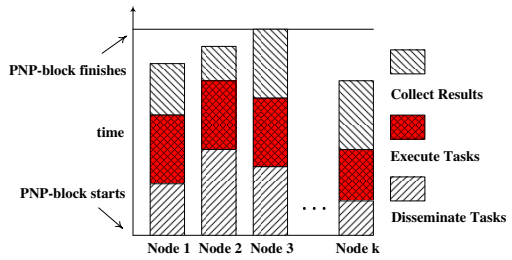


Figure 4: The PNP-block completion time is composed of a) the time to disseminate tasks, b) the time to execute tasks and c) the time to collect results, in addition to the time to execute pre-process and post-process programs.

input, and its output. The number of tasks will be taken as a parameter for the pre-process program. The input can be the original input or an intermediate output of its parent PNP-blocks. Every directed link of the job DAG should be declared by specifying the parent PNP-block and children PNP-block. The job parser also conducts some simple checking, for instance, checking for the existence of cycles.

3.4 Master

The master is responsible for starting, monitoring, swapping and terminating the workers. When a worker finishes a task, the master will send a request to the job engine for a new task. During the execution process, the master monitors the worker status. If an exception is captured, the master will terminate the worker and send a report to the job initiator. The master may swap out the workers when the device is busy with other applications.

4. TASK ALLOCATION FOR PNP-BLOCKS

PNP-blocks are the basic blocks to allocate tasks in Serendipity. The communication between two connected PNP-blocks of a job resides in the same node (i.e., its job initiator), making the communication time negligible. Therefore, the entire job completion time is determined by the sum of all the PNP-block completion times on the critical path of the job DAG. Efficient task allocation algorithms will reduce the PNP-block completion time and, thus, the entire job completion time. In this section, we will focus on the task allocation for PNP-blocks.

Figure 4 illustrates the timing and components of a PNP-block execution. Along the x-axis are the k remote nodes that will execute the parallel tasks of the block. Along the y-axis is a depiction of the time taken at each remote node to receive disseminated tasks from the initiator, execute those tasks, and provide the result collection back to the initiator. As illustrated, the time for each remote node to receive its disseminated tasks may vary, depending on the availability and quality of the network between the initiator and the remote node. When n tasks of a PNP-block are allocated to k nodes, each node will execute its assigned tasks sequentially, again generally taking a variable amount of time. After execution of all assigned tasks in the block, the node will send results back to the initiator, with time again dependent on the network between the initiator and the remote node. The PNP-block completion time depends on the time the last result is received by the job initiator. Our basic goal for task allocation is to reduce the completion time of the last task. Based on the properties of various DTNs discussed in Section 2.1, we design three task allocation algorithms for these DTNs.

4.1 Predictable DTNs with Control Channel

We first consider an ideal DTN setting where the future contacts can be accurately predicted, and a control channel is available for information sharing. The performance in this kind of DTN represents the best possible performance task allocation that is achievable in DTNs.

With future contact information a Dijkstra's routing algorithm for DTNs [21] can be used to compute the required data transfer time between any pair of DTN nodes given its starting time. With the control channel the job initiator can obtain the time and number of tasks to be executed on the target node with which to estimate the time to execute a task on that node. Therefore, given the starting time and the target node, the task completion time can be accurately estimated.

Using this information, we propose a greedy task allocation algorithm, *WaterFilling*, that iteratively chooses the destination node for every task with the minimum task completion time (see Algorithm 1).

Algorithm 1 Water Filling

```
1: procedure WATERFILLING(T, N)   ▷ T is the task set; N is node set.
2:     current ← currentTime();
3:     rsv ← getTaskReservationInfo();
4:     inputSize ← getTaskInputSize(T);
5:     outputsize ← estimateOutputSize(T);
6:     queue ← initPriorityQueue();
7:     for all n ∈ N do
8:         arrivalTime ← dijkstra(this, n, current, inputSize);
9:         exeTime ← estimateTaskExecutionTime(n,t);          ▷ t ∈ T
10:        tfinishTime ← taskFinishTime(rsv[n], arrivalTime, exeTime);
11:        completeTime ← dijkstra( n, this, tfinishTime, outputSize);
12:        queue.put({n, arrivalTime, exeTime, completeTime});   ▷
   Nodes are sorted according to their completeTime.
13:    end for
14:    for all t ∈ T do
15:        {n, arrivalTime, exeTime, receiveTime} ← queue.poll();
16:        updateReservation(rsv[n], t, inputSize, arrivalTime, exeTime);
   ▷ update both network usage and execution time for future computing.
17:        send(n, t);
18:        arrivalTime ← dijkstra(this, n, current, inputSize);
19:        tfinishTime ← taskFinishTime(rsv(n), arrivalTime,exeTime);
20:        completeTime ← dijkstra( n, this, tfinishTime, resultSize);
21:        queue.put({n, arrivalTime, exeTime, receiveTime});
22:    end for
23:    reserveTaskTime(rsv);     ▷ inform all nodes of the allocated tasks
24: end procedure
```

For every task, the algorithm first estimates its task dissemination time to every node. With the information of the tasks to be executed on the destination node and the estimated time to execute this task, it is able to estimate the time when this task will finish. Given that time point, the time

when the output is sent back can also be computed. Among all the possible choices, we choose the node achieving minimal task completion time to allocate the task. The allocation of the next task will take the current task into account and repeat the same process. Finally, the job initiator will reserve the task execution time on all related nodes, which will be shared with other job initiators for future task allocation.

Algorithm 2 Compute on Dissemination

```
1: procedure ENCOUNTER(n)                          ▷ n is the encountered node.
2:     summary ← getSummary();
3:     send(n, summary);
4: end procedure
5: procedure GETSUMMARY
6:     compute ← getNodeComputingSummary();
7:     net ← getNetworkSummary();
8:     tasks ← getPendingTaskSummary();
9:     return {compute,net,tasks};
10: end procedure
11: procedure RECEIVESUMMARY(n, msg)                ▷ msg is the summary
    message of node n.
12:     updateNodes(msg.compute);
13:     updateNetwork(msg.net);
14:     toExchange ← exchangeTask(n, this.tasks, msg.tasks);
15:     isSent ← false;
16:     while n.isConnected() && !toExchange.isEmpty() do
17:         send(n, toExchange.poll());
18:         isSent ← true;
19:     end while
20:     if n.isConnected() && isSent == true then
21:         summary ← getSummary();
22:         send(n, summary);
23:     end if
24: end procedure
25: procedure RECEIVETASK(msg)                      ▷ msg contains exchanged tasks.
26:     addTasks(msg.tasks);
27: end procedure
```

There are several practical issues in implementation of the algorithm. First, the algorithm needs to estimate the time to execute tasks. Local execution time can be estimated by running several tasks on local node since every task of a PNP-block have similar execution time. By comparing the device description input by users or historical information of task execution time, the task execution time on other nodes can also be estimated. Second, to accurately compute the data transfer time, an advanced version of Dijkstra's algorithm for DTNs requires every DTN bundle to reserve the transfer time on all node along its routing path to avoid contention with other bundles, which will overload the low-capacity control channel and, thus, is impractical. Our implementation uses the simple version of Dijkstra's algorithm that does not consider the effect of other bundle. It will cause long delay when the network is congested. Third, since the task execution time and task arrival time may be inaccurate, tasks may fail to finish execution within their reserved execution duration. To solve this problem, all tasks are sorted according to their reserved time in the task list. Tasks arriving late will probably be executed immediately after arrival.

4.2 Predictable DTNs without Control Channel

When DTNs have no control channels, it is impossible to reserve task execution time in advance. WaterFilling will cause contention for task execution among different jobs on popular nodes which have high contact frequencies with other nodes, prolonging the task execution time. To solve this problem, we propose an algorithm framework, Compute on Dissemination (CoD), to allocate tasks in an opportunistic way. The algorithm is shown in Algorithm 2.

The basic idea of CoD is that during the task dissemination process, every intermediate node can execute these tasks. Instead of explicitly assigning a destination node to every task, CoD opportunistically disseminates the tasks among those encountered nodes until all tasks finish. Every time two nodes encounter each other, they first exchange metadata about their status, e.g., device information, network conditions, and pending tasks. Based on this information, they decide the set of tasks to exchange. When they move out of the communication range, they will keep the remaining tasks to execute locally or exchange with other encoutered nodes in the future.

Algorithm 3 Task Exchange in Predictable DTNs

```
1: procedure EXCHANGETASK(n, localT, remoteT)      ▷
   n is the encountered nodes. localT and remoteT are the tasks on the
   local and remote nodes, respectively.
2:     current ← currentTime();
3:     localTaskTime ← 0;
4:     remoteTaskTime ← estimateTotalExecutionTime(remoteT);
5:     for all t ∈ localT do
6:         exeTime ← estimateTaskExecutionTime(this, t);
7:         remoteExeTime ← estimateTaskExecutionTime(n,t);
8:         size ← estimateOutputSize(t);
9:         localFinish ← current + localTaskTime + exeTime;
10:        remoteFinish ← current + remoteTaskTime + remoteExeTime;
11:        localComplete ← dijkstra(this, t.initiator, localFinish, size);
12:        remoteComplete ← dijkstra(n, t.initiator, remoteFinish, size);
13:        if localComplete > remoteComplete then
14:            toExchange.add(t);
15:            localT.remove(t);
16:            remoteTaskTime ← remoteTaskTime + remoteExeTime;
17:        else
18:            localTaskTime ← localTaskTime + exeTime;
19:        end if
20:    end for
       return toExchange;
21: end procedure
```

The key method of this algorithm framework is to decide which tasks to exchange. In this subsection we consider the DTNs whose future contact is still predictable. In such DTNs the task completion time can be estimated when the task arrives at a node as discussed in last subsection. The intuition of CoD in predictable DTNs (pCoD) is to locally minimize the task completion time of every task if possible. When two nodes encounter, they will compare the task completion time of every task to be executed locally and that to be executed remotely. If the local task completion time is longer than the remote one, the node will send the task to the other node. Algorithm 3 shows the details.

Both nodes may send tasks of various jobs to each other

to reduce their completion time. Every node conservatively makes the decision without considering the tasks the other node will send back. It is reasonable because of the uncertainty of network usage. If they are still within communication range when their chosen tasks have been exchanged, they will exchange a summary of current task list, which may trigger a new round of task exchange. If they always stay connected, the task exchange process will finally converge.

4.3 Unpredictable DTNs

In this subsection we consider the general case of an unpredictable DTN, a DTN where the future contacts cannot be accurately predicted. Since it is meaningless to use a control channel to reserve task execution time in such DTNs, we will not treat unpredictable DTNs with a control channel separately.

Our task allocation algorithm, CoD for unpredictable DTNs (upCoD), is still based on CoD with the constraint that future contact information is unavailable. As shown in Figure 4, minimizing the time when the last task is sent back to the job initiator will reduce the PNP-block completion time. When the data transfer time is unpredictable, we envision that reducing the execution time of the last task will also help reduce PNP-block completion time. Therefore, when two nodes encounter each other, upCoD tries to reduce the execution time of every task. Algorithm 4 shows the details.

Algorithm 4 Task Exchange in Unpredictable DTNs

```
1: procedure EXCHANGETASK(n, localT, remoteT)        ▷ n is the encountered node. localT and remoteT are the tasks on the local and remote nodes, respectively.
2:     current ← currentTime();
3:     localTaskTime ← 0;
4:     remoteTaskTime ← estimateTotalExecutionTime(remoteT);
5:     for all t ∈ localT do
6:         exeTime ← estimateTaskExecutionTime(this, t);
7:         remoteExeTime ← estimateTaskExecutionTime(n,t);
8:         localFinish ← current + localTaskTime + exeTime;
9:         remoteFinish ← current + remoteTaskTime + remoteExeTime;
10:        if localFinish > remoteFinish then
11:            toExchange.add(t);
12:            localT.remove(t);
13:            remoteTaskTime ← remoteTaskTime + remoteExeTime;
14:        else
15:            localTaskTime ← localTaskTime + exeTime;
16:        end if
17:    end for
       return toExchange;
18: end procedure
```

In unpredictable DTNs historical contact information may be helpful to task exchange in CoD. We will investigate such possibility as part of our future work.

5. PNP-BLOCK SCHEDULING

Our PNP-block design simplifies the task allocation so that every PNP-block is treated independently. However, it is still possible to further reduce the job completion time by assigning priorities to PNP-blocks since tasks from the same job are executed according to their priority assignment.

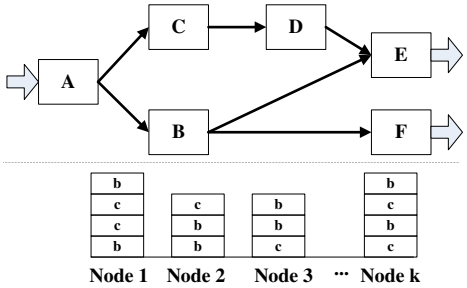


Figure 5: A job example where both PNP-block B and C are disseminated to Serendipity nodes after A completes. Their task positions in the nodes' task lists are shown blow the DAG.

The key to reduce the job completion time is to minimize the total time along the critical path of the job DAG. Unfortunately, the PNP-block completion time is affected by many factors, which may change due to some random events. The critical path of a job DAG is, therefore, hard to identify, if not impossible. To solve this problem, we propose some heuristics for priority assignment based on our observation of the DAG structure.

As shown in Figure 5, it is possible that two PNP-blocks of a job are simultaneously disseminated, for example PNP-blocks B and C in the same figure. Their tasks arrive at the destination nodes unordered. Given a DTN and a task allocation algorithm, the total time required for both B and C to finish remains almost the same. However, either B or C can have a shorter PNP-block finish time if any of them is given a higher priority over the other. This will be beneficial because their children PNP-block can start earlier.

OBSERVATION 1. *It is always better to assign different priorities to the PNP-blocks of a job than to assign the same priority to them.*

In the example shown in Figure 5, PNP-block E can only start when both B and D finish. Thus, B and D are equivalently important to PNP-block E. Meanwhile, there is a time gap between the execution time of PNP-block C and that of PNP-block D caused by the result collection of C and task dissemination of D. During that gap, the execution of other tasks (e.g., B) will not affect the PNP-block finish time of D. Therefore, if C is assigned a higher priority than B, the total time for both B and D to finish will be shorter.

OBSERVATION 2. *All parents of a PNP-block are equivalently important to it, while parents should have higher priorities than their children.*

The next question arises when B and D are in the task list of the same node, which should have higher priority. We notice that both B and D are equivalent to E, while E and F are equivalent to the entire job. However, if B finishes earlier, F can start earlier. This is because F only relies on B.

OBSERVATION 3. *When two PNP-blocks have the same priority, the one with more children only depending on it should be assigned a higher priority.*

Algorithm 5 PNP-block Priority Assigning

```
1: procedure ASSIGNPRIORITY(J)                          ▷ J is the job DAG
2:     while !J.allPNPblocksHavePriority() do
3:         for all s ∈ J do                              ▷ s is a PNP-block
4:             if !s.haveChild() then
5:                 s.priority ← 0;
6:             else if s.allChildrenHavePriority() then
7:                 s.priority ← s.maxChildrenPriority()+1;
8:             end if
9:         end for
10:    end while
11:    for p = 0 → J.getMaxPriority() do
12:        PNPblocks ← J.getPNPblocksWithPriority(p);
13:        sort(PNPblocks);
14:        for i = 0 → PNPblocks.size()-1 do
15:            s ← PNPblocks.get(i);
16:            s.priority ← s.priority + i / PNPblocks.size();
17:        end for
18:    end for
19: end procedure
```

If there are still PNP-blocks with the same priority, we randomly assign some different priorities to them that keep their relative priorities with other PNP-blocks. Algorithm 5 shows our priority assigning algorithm. The sort method of line 13 is based on Observation 3.

6. SYSTEM IMPLEMENTATION

In this section we provide more details on the system implementation aspects of Serendipity related to its robustness against failures.

6.1 Failure Recovery

Serendipity expects errors and failures to occur during the job execution. There are generally two types of failures, namely job failure and platform failure.

There are various possible errors in a job that result in job failures. For example, the program may have bugs or the input data may contain errors. We notice that the results of finished PNP-blocks are reusable if their corresponding programs have no bug. When the failure is caused by the input data, still a large portion of the tasks in the current PNP-block have correct results. We allow the user to reuse a part of these results by storing checkpoints for every completed PNP-block. When a pre-process, a post-process or a task program fails, the worker and the master will try to capture the exceptions and report them to the job initiator. On receiving the report, the job initiator will immediately report the current job progress and the exceptions to the user. Meanwhile, it will continue collecting the results of the executed PNP-blocks. After all results and exception reports are collected, the job initiator will record the last checkpoint and stop. After fixing the errors, the user can request the job engine to resume the updated job from any of the stored checkpoints.

The platform failure type includes job initiator failures and worker failures. This happens when, for example, the device owners turn off the devices. If the node that has been assigned the job initiator fails, the entire job fails. Fortunately, since the job initiator records checkpoints during the execution of the job, it can continue the job when the Serendipity node restarts again. The job engine will check all the incomplete jobs and ask their corresponding users if these jobs should continue. If the user chooses to continue, the job engine will restart the corresponding job initiators based on their latest checkpoint. It is also possible that the worker node fails. Serendipity recovers from this kind of failures through task replication. More details about the task replication are discussed in the following subsection.

6.2 Task Replication

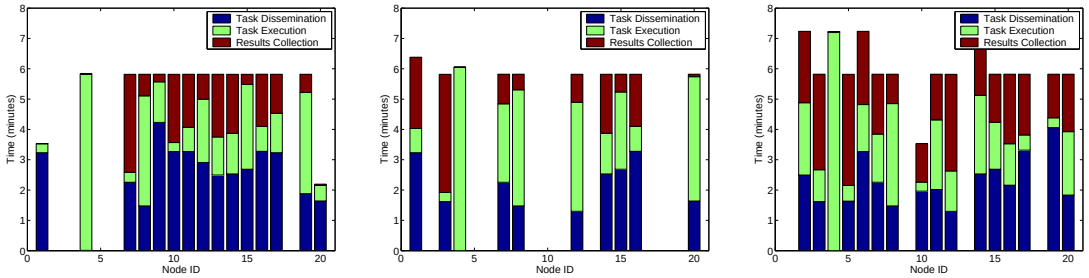
Serendipity relies on task replication to recover from worker failures. Task replication also helps reduce the job completion time when tasks experience unexpected long delays. Serendipity adapts to passive task replication strategies based on the consideration of reducing the power usage of the DTN nodes. There are several strategies available to be configured in the configure file. The first strategy will randomly choose an unfinished task to run on the local device when it is idle. It is suitable for the scenario where the local node has enough power and wants to finish the job as soon as possible. The second strategy will gradually replicate the remaining tasks locally or remotely after the latest task results have been received for a certain period of time. The time threshold can be dynamically set based on the the task execution time and DTN settings. In the current implementation, Serendipity uses the first task replication strategy.

7. EVALUATION

7.1 Experiment Setup

Testbed: We have built a DTN testbed on Emulab [35]. Every DTN node that runs on an Emulab node has an emulation module for the DTN physical layer, besides the common DTN modules. Before an experiment starts, all DTN nodes load the contact traces in their emulation modules. The contact traces can be real DTN traces extracted from humans or vehicles (e.g., Haggle[18], DieselNet[8]) or can be generated according to some mobility models (e.g., Random WayPoint model[31]). When a contact occurs according to the loaded traces at some time point, the emulation modules of the corresponding nodes will establish a UDP connection with each other and notify their DTN nodes. From then onwards these DTN nodes can use this connection to transfer data until the contact ends. The connection bandwidth can either be specified in the contact traces or configured in the configure file.

Mobility Models: We use a set of mobility models to generate contact traces, namely Random WayPoint Model (RWP) [31], Truncated Levy Walk Model (TLW) [29], Time-Variant Community Mobility Model (TVCM) [22], and Manhattan Model [4]. These models represent a wide range of mobility patterns. RWP is the simplest model and assumes unrestricted node movement. TLW describes the human walk



(a) WaterFilling (b) pCoD (c) upCoD

Figure 6: The composition of the job completion time for the base case with various task allocation algorithms. Node 4 is the job initiator.

pattern verified by collected mobility traces. TVCM depicts human behavior in the presence of communities. Manhattan model represents a mobility pattern where node movement is restricted by the underlying road maps. The basic settings assume a 1 Km by 1 Km square activity area. Each node has a 100 m diameter circular communication range. The remaining mobility model parameters are varied across the different experiments to analyze their effect on the performance of Serendipity.

Performance Metrics: We consider the following performance metrics:

- *Speedup:* The ratio of the job completion time on a single device to that on the whole Serendipity platform is the primary metric to evaluate the performance of Serendipity. We define the sum of all task execution times as the job completion time on a single device, which might be slightly shorter than its real value. The job completion time of Serendipity is equal to the time elapsed from when a job is submitted to when the job completes. The larger speedup values reflect better performance.
- *Network traffic:* Because of the scarcity of communication opportunities in DTNs, the amount of traffic generated by Serendipity will also be considered.

Applications: We implement a speech-to-text application based on SPHINX [23] that translates audio to text as might be helpful in a developing world scenario such as described in the Introduction. It is a computationally complex application that takes long time to execute on a single mobile device. We have implemented it as a single PNP-block job where the pre-process program divides a large audio file into multiple pieces, each of which is the task input.

Baseline: To analyze the importance of task allocation in DTNs, we implement an encounter-based task allocation algorithm, named *Encounter*, where the job initiator allocates a task to an encountered node only when the latter is idle. In other words, the task allocation scheme only relies on the contact opportunities. As for priority assignment, we compare our algorithms against the case of basic Serendipity without priority assignment.

Expriment Settings: All of our experiments are conducted on the Emulab-based DTN testbed. Every machine has a 600 MHz Pentium III processor and 256 MB memory,

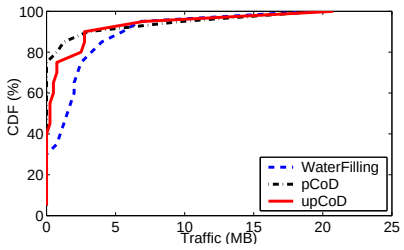


Figure 7: CDF of the traffic that each node transmits.

which is less powerful than mainstream PCs but closer to that of smart mobile devices. Every experiment is repeated three times with different seeds[2]. The results reported correspond to the average values.

7.2 Experiment results

7.2.1 A Base Case

We initiate the experiments with the speech-to-text application using three task allocation algorithms. The size of the audio file is 25 MB. As mentioned before, it is implemented as a single PNP-block job whose pre-process program divides the audio file into 100 pieces corresponding to 100 tasks. Its post-process program collects and combines the results. There are 20 nodes randomly distributed in the activity area. Their movement follows RWP with an average speed of 5 meters per second (m/s). The wireless bandwidth is assumed to be 2 Mbps. The owner of node 4 starts the job at time 0.

Figure 6 demonstrates the composition of the job completion time with the three task allocation algorithms. As expected, WaterFilling achieves the best performance, 5.9 minutes, since it is able to globally optimize the job completion time. The job completion time of pCoD is 6.5 minutes, while that of upCoD is 7.3 minutes. The execution time of every task is around 15 seconds. The speedups in this case are, therefore, 4.2, 4.0 and 3.4 for WaterFilling, pCoD and upCoD, respectively.

The task dissemination time distributions, the task execution time and the result collection time demonstrate some

[2]These experiments are very time-consuming. We argue that the average values of three experiments are enough to show the trends. For any of the experiments that we repeated ten times we obtained very similar values.

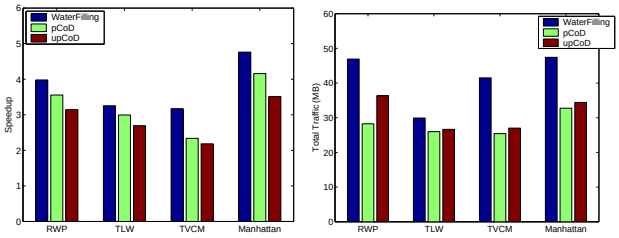
interesting phenomena reflecting the properties of these algorithms. First, the task dissemination time of WaterFilling is longer than those of pCoD and upCoD. This happens because with WaterFilling the tasks a node receives during its first few contacts with the job initiator are for nodes far way at that time point from the job initiator. On the contrary, a pCoD and a upCoD node will immediately start a task once it receives it. Second, upCoD has the largest number of nodes executing tasks because it tries to disseminate tasks out when two nodes encounter. As opposed to upCod, pCoD may reject task dissemination opportunities without considering the encountered node as an intermediate node. Therefore, its number of nodes executing tasks is much smaller than that of WaterFilling and upCoD. These properties indicate that WaterFilling leads to the largest amount of traffic transmitted in the network, while pCoD causes the least traffic. This is verified by the traffic distributions shown in Figure 7.

With WaterFilling 65% of the nodes send at least 1 MB of data, while those for pCoD and upCoD are 20% and 25%, respectively. However, the amount of traffic sent by the job initiator is similar for all three algorithms. It is also worth noting that the job initiator sends the most traffic among all nodes.

7.2.2 The Impact of the Network Environments

Next we analyze the impact of network environments on the performance of the three task allocation algorithms by changing the network settings in the base case.

Mobility Models: We first compare the impact of different mobility models. We use RWP, TLW, TVCM and Manhattan models to represent a wide range of mobility patterns. In these experiments, we set the node speed to 5 m/s. For the Manhattan model, we use three by three streets. All other parameters are the default settings provided in the respective papers.

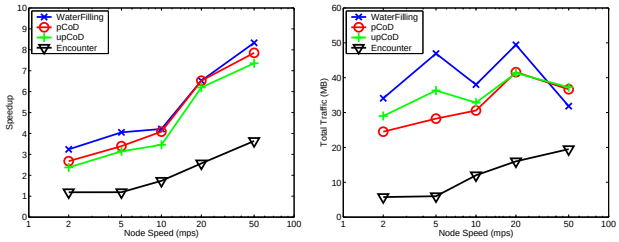


(a) Speedup (b) Total network traffic

Figure 8: The performance of Serendipity with various mobility models.

The results of this comparison are shown in Figure 8. In all these models, WaterFilling obtains the highest speedup while introducing the most traffic. pCoD performs better than upCoD in terms of both speedup and network traffic in all experiments. Among all these models, Serendipity achieves highest speedup in Manhattan model, namely 4.8, 4.2 and 3.5 for WaterFilling, pCoD and upCoD, respectively. The reason for this is that nodes attain higher contact frequency when they move on the restricted roads. The gap between WaterFilling and pCoD / upCoD is the largest in TVCM, indicating that the local optimization may fail to find some task dissemination paths without global knowledge. This happens because among the multiple communities in TVCM there are a few long path that are hard to identify with only local information. pCoD and upCoD exhibit similar network traffic with TLW, TVCM and Manhattan.

Node Speed: Node speed affects the contact frequencies and the contact durations, which are critical to DTNs. We change the node speed from 2 m/s, i.e., human walking speed, to 50 m/s, i.e., an extremely high speed. We use Encounter as the baseline since it relies on frequent contacts to disseminate tasks.



(a) Speedup (b) Total network traffic

Figure 9: The impact of node speed on Serendipity.

The results are demonstrated in Figure 9. Our algorithms perform significantly better than the baseline. Regardless of the task allocation algorithm, Serendipity's performance increases with the node speed. WaterFill still performs best for all speeds. However, with the increase of the speed, the speedup difference of three task allocation algorithms decreases. When the speed is lower than 10 m/s, the speedup of pCoD is close to that of upCoD. But when the speed exceeds 10 m/s, its speedup is closer to that of WaterFilling. This indicates that, with high contact frequencies, local optimization manages to approximate the global optimization. In fact, when the speed reaches 50 m/s, even Encounter speeds up the job by a factor of 3.6.

As shown in Figure 9(b), the total network traffic varies with the speed. It drops when the speed increases from 5 m/s to 10 m/s, and grows again when the speed increases to 20 m/s. To analyze the reason, we plot the traffic distribution for WaterFilling in Figure 10. When the speed is 5 m/s, a small portion of nodes act as intermediate nodes and transmit a lot of data. When the speed increases to 10 m/s, these nodes don't need to deliver data for the job initiator because it is more efficient for the job initiator to disseminate the tasks with its increasing contact frequency. However, when the speed further increases to 20 m/s, most nodes transmit more data for the job initiator, increasing once again the total traffic.

Number of Nodes: We next examine how the quantity of available resources affect the performance of Serendipity. There are two possible scenarios; the node density either increases with the number of nodes or remains the same. In order to to show the effects, we change the number of nodes in the base case. By keeping the activity area fixed, the node

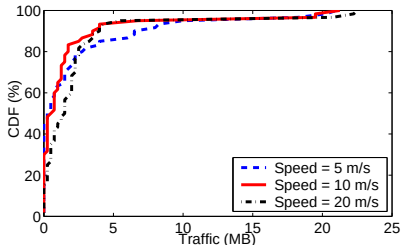
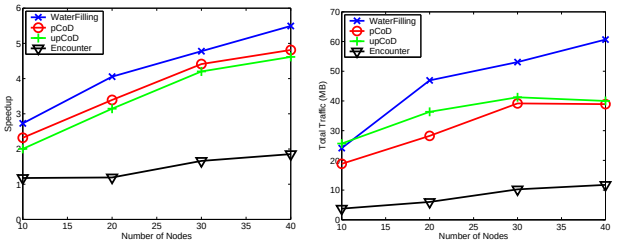


Figure 10: The traffic distribution of Serendipity with various node speeds. WaterFilling is used for task allocation.

density changes with the number of nodes. Figure 11 shows the results when the number of nodes varies from 10 to 40.

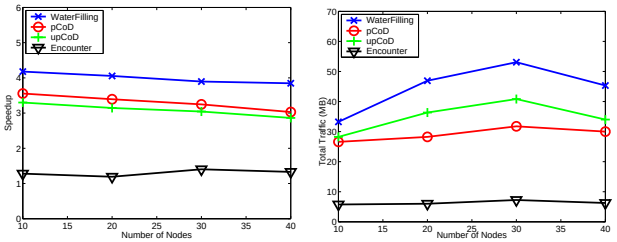


(a) Speedup (b) Total network traffic

Figure 11: The impact of node numbers on Serendipity. The activity area is fixed.

With the increase in number of nodes, the speedups of the three task allocation algorithms increase from 2.7, 2.3, and 2.0 to 5.5, 4.8, and 4.6 for WaterFilling, pCoD and upCoD, respectively. The baseline, Encounter, also increases from 1.2 to 1.9. The relative performance of the three algorithms does not change with the number of nodes. Meanwhile, as shown in Figure 11(b), the total traffic of upCoD is higher than that of pCoD when the number of nodes is less than 30. When it increases to 30, their total traffic is similar to each other. This happen because the amount of traffic exchanged among nodes is also restricted by the number of tasks for upCoD.

It is also of interest to understand the impact of node density on the performance of Serendipity. We proportionally change the activity area with the number of nodes so that the node density remains constant. Figure 12 shows the results.

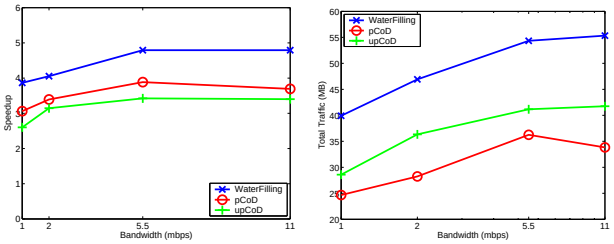


(a) Speedup (b) Total network traffic

Figure 12: The impact of node numbers on Serendipity. The node density is fixed, i.e., the activity area changes with the number of nodes.

The speedup drops slightly with the increase of the number of nodes for all the aforementioned task allocation algorithms. This comes as a result of the increased round trip times between two nodes due to the increase of the activity area. When disseminating the same number of tasks, the job completion time will slightly increase accordingly. It is also interesting that the total traffic of WaterFilling and upCoD decreases when the number of nodes increases from 30 to 40, as shown in Figure 12(b). Their value is closer to that of pCoD when there are 40 nodes. This happens because the increase in the number of nodes makes the role of intermediate nodes in task dissemination less important.

Bandwidth: Finally, we consider the effect of wireless bandwidth on the performance of Serendipity. The bandwidth is set to 1 Mbps, 2 Mbps, 5.5 Mbps and 11 Mbps, which are typical values for wireless links. Since Encounter primarily relies on the contact opportunities to disseminate tasks, it will not be affected much by the bandwidth. Figure 13 plots the results.



(a) Speedup (b) Total network traffic

Figure 13: The impact of wireless bandwidth on Serendipity.

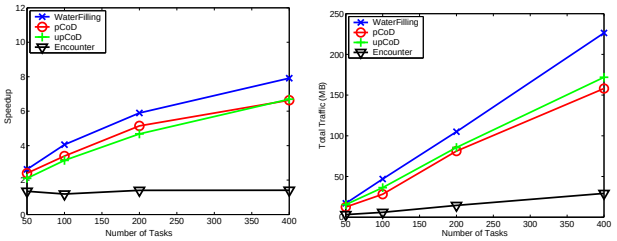
When the bandwidth increases from 1 Mbps to 5.5 Mbps, the speedups increase from 3.9, 3.1 and 2.6 to 4.8, 3.9, and 3.4 for WaterFilling, pCoD and upCoD, respectively. The relative increases are 23.8%, 27.1% and 31.5%. Increasing the bandwidth helps deliver more data on every contact opportunity, speeding up the task dissemination process. The greedy nature of pCoD and upCoD helps them benefit more when more resources are available. However, when the bandwidth increases from 5.5 Mbps to 11 Mbps, the speedup does not change much for all three algorithms. pCoD even slightly decreases to 3.7, indicating that when the bandwidth is large enough, the availability of contact opportunities plays the key role on the performance of Serendipity. The decrease of pCoD is probably due to its greedy strategy. As more data can be transferred per contact opportunity, some tasks are trapped by local optimal nodes, increasing the total job completion time. The total network traffic consistently changes with their corresponding speedup, verifying the important role bandwidth plays in task dissemination.

7.2.3 The Impact of the Job Properties

Next we evaluate how the job properties affect the performance of Serendipity.

Job Size: First, we consider changing the input data and, consequently, the number of tasks for the same applications. Figure 14 depicts the results when task number changes from 50 to 400, corresponding to 12.5 MB and 100 MB audio files.

The speedups increase from 2.6, 2.4, and 2.1 to 7.9, 6.7



(a) Speedup (b) Total network traffic

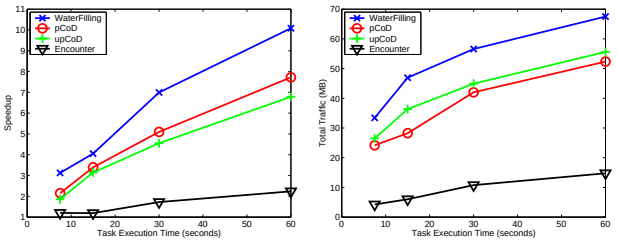
Figure 14: The impact of job size on Serendipity.

and 6.7 for WaterFilling, pCoD and upCoD, respectively. In other words, the speedups almost increase by 200% for all algorithms. In contrast, the performance of Encounter does not change considerably. The better performance with larger job size is because Serendipity simultaneously disseminates some tasks and executes other arrival tasks. It does not increase the task dissemination time for the entire job but increases the utilization of system. The reason why the performance does not linearly increase with the job size is because it also takes time to disseminate tasks and collect results. In contrast, the total traffic almost linearly increases with the job size for all three algorithms.

The results show that for the DTN environment the additional delay incurred by intermittent contacts is not severe enough for large full-parallel jobs. Increasing the parallelism will help increase the performance of Serendipity.

Task Complexity: Another property of the jobs is the task complexity. Given the same job structure and input data, if every task requires more time to execute, the ratio of task execution time to the job completion time will increase. As a result, the system utilization and the job speedup will increase.

To verify our intuition, we change the task computing time by adding dummy computations to the speech-to-text application or substituting the real computation with dummy computation when the desired computing complexity is lower than the real application. Figure 15 demonstrates the results.



(a) Speedup (b) Total network traffic

Figure 15: The impact of task complexity on Serendipity.

When the task execution time increases from 7.5 seconds that corresponds to half of the real task execution time of the speech-to-text application to 60 seconds, the speedup increases from 3.1, 2.2, and 1.9 to 10.1, 7.7, and 6.9 for WaterFilling, pCoD and upCoD, respectively. While the total task execution time corresponds that in the last set of experiments, the speedups are higher than those of the above experiments because this set of experiments has less data to disseminate out when the computing complexity is high. Thus, the contact opportunities can be used better by distributing the tasks out via intermediate nodes. As shown in Figure 15(b), the total traffic increases with task execution time although the total input data of the job does not change. The increased traffic is caused by transmitting data via intermediate nodes.

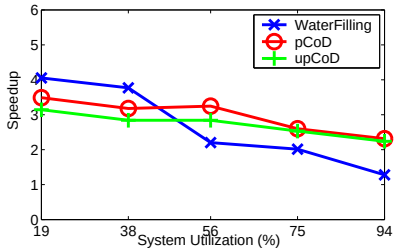


Figure 16: The performance of different task allocation algorithms with multiple jobs.

Multiple Jobs: A more practical scenario involves DTN nodes submitting multiple jobs simultaneously into Serendipity. These jobs will affect the performance of each other when their execution duration overlaps. In this set of experiments, nodes will randomly submit jobs into Serendipity. The arrival time of these jobs follows Poisson distribution. We change the arrival rate so that the system utilization varies from 19% (low) to 94% (very high). Figure 16 shows the results.

As expected, the speedup decreases when the system utilization increases. However, we should note some interesting phenomena. First, WaterFilling performs better than pCoD and upCoD when the system utilization is relatively low, i.e, 38%. When it increases to 56%, its speedup drastically drops to 2.2. When the system utilization reaches 94%, its speedup is only 1.28. This happens because the contention for contact opportunities among different jobs causes unexpected long delays when the job arrival rate is high. In contrast, the local optimization strategies of pCoD and upCoD make them perform better when there are a lot of jobs. They achieve 3.3 and 2.8 speedup when the system utilization is 56% and 2.3 and 2.2 speedup even when the system is almost fully used. pCoD outperforms upCoD when the system utilization is lower than 75%. With the further increase of job arrival rate, they achieve similar performance.

This set of experiments demonstrates the advantage of distributed computing in Serendipity. Even at times when all nodes are busy with tasks, it is still beneficial to execute tasks in a distributed way, which exhibits lower job completion time.

7.2.4 DAG jobs

The above experiments show that Serendipity performs well for single PNP-block jobs. Since DAG jobs are executed iteratively for all dependent PNP-blocks while parallel for all independent PNP-blocks. The above experiment results also apply to DAG jobs. In this set of experiments we

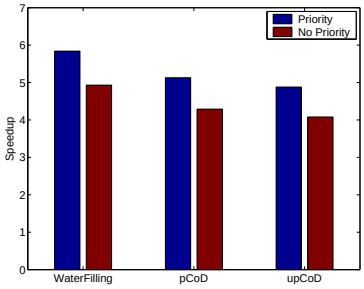


Figure 17: The importance of assigning priorities to PNP-blocks.

will evaluate how PNP-block Scheduling algorithm further improves the performance of Serendipity.

We use the job structure shown in Figure 5, as might reflect the situational awareness scenario in the Introduction where the processing of one image impacts the processing of another. to use the PNP-blocks of speech-to-text application as the basic building blocks. PNP-block A has 0 tasks; PNP-block B has 200 tasks; PNP-block C has 50 tasks; PNP-block D has 100 tasks; PNP-block E has 100 tasks; PNP-block F has 0 tasks. The performance difference between our algorithm and assigning equal priority to the PNP-blocks is shown in Figure 17.

Our priority assignment algorithm achieves the speedup of 5.8, 5.1 and 4.9 for WaterFilling, pCoD, and upCoD, consistently outperforming that of 4.9, 4.3, and 4.1 when all PNP-blocks have the same priority. This set of experiments demonstrates the usefulness of priority assigning. Further evaluation of our algorithm on diverse type of jobs will be part of our future work. It's also worth noting that the priority assignment algorithm does not change the relative performance among three task allocation algorithms because these three algorithms target at PNP-blocks, the fundamental blocks of jobs. Therefore, jobs with various DAG structures will all benefit from good task allocation algorithms for PNP-blocks.

8. RELATED WORK

Serendipity has aspects in common with prior work in applications in DTNs, distributed computing in non-dedicated and volunteer settings over generally well-connected wired networks, and in mobile distributed computing over wireless networks. We consider each of these in turn.

Applications in DTNs: In the developing world, the DakNet [28] and KioskNet [16] projects stand out for their success in achieving deployment and bringing applications such as email, voicemail, and asynchronous web page access to regions without Internet or local intranet services. Architecturally, both projects make use of mobile access points to provide asynchronous connectivity between rural village kiosks and the Internet accessed in a city. In each of these projects, the network is used for transport, not for computation.

In the disaster response area, Hanna et al. have considered mobile distributed information retrieval systems in a pure DTN context [17]. This work considers an application-level service that allows sharing of documents in highly-partitioned networks, focusing on the key issue of replication strategies with a simple neighbor-search approach to finding documents. We share an emphasis on the pure DTN environment and the provision of a timely service. Our work differs in that we consider services that require computation, not just information search and delivery.

Closer to our work is the recent architecture proposed for disaster communications response by Fall et al. [15] with a specific focus on situational awareness. The authors propose an architecture that contains infrastructure-supported back-end servers, mobile producer/consumer nodes and mobile field servers, all running the DTN protocols for communication. The architecture provides for content-based forwarding as an augmentation of basic DTN forwarding, as a mechanism to reduce redundant traffic. This project is relatively new; system prototyping and integration of computation such as natural language processing, object recognition, and image compression are intended for future work. Related, the Hastily Formed Networks (HFN) project [13] describes potential applications in disaster settings that match well with our vision requiring computation, including situational awareness, information sharing, planning and decision making. HFN has a highly practical focus on providing technologies for immediate use in natural and man-made disasters.

In the military setting, DARPA has invested heavily in the development of DTN protocols and demonstration projects as well as follow on Wireless Network After Next (WNAN) development. An overview of the progress and vision for military use can be found in Marshall [26], though this work is rather light in specifying desired applications. Still the Marshall paper and WNAN program announcement do anticipate that "the network will create a distributed computing environment, where the applications and services are populated/migrated onto nodes according to traffic flows and resource availability'". DTNs have also been examined for military use in naval networks [30] and in the Marine Corps CONDOR system [27], though the public descriptions of these systems emphasize communication over computation-oriented applications.

Distributed computing on non-dedicated machines: In the area of distributed computing, our work is most closely related to systems that use non-dedicated machines with cycles that are donated and may disappear at any time. In this vein, our work takes some inspiration from the Condor system for the job model and master/worker system architecture [24]. Condor focuses on ease-of-use for job submission and result completion, providing an interface that requires no code modification and including checkpointing to recover from any early terminations. Condor also provides the ability for resources to be added to the system with a specification of the constraints and characteristics. Some of these features may be useful in our setting as our project matures.

From the standpoint of completing a distributed computa-

tion, a network that prevents timely delivery of results looks for some purposes indistinguishable from a node that stops computing. With this observation, our system resembles in part those distributed computing environments that have well-connected networks but flaky participation in the computation, such as those seen in voluntary computing efforts where users can contribute compute cycles, but may also simply turn off their machines or networks at will in the middle of a computation. Examples of these systems include BOINC [2]; other examples are SETI@home [3], and folding@home[7], all leveraging willingness on the part of individuals to dedicate resources to a large computation problem. Like these systems, we provide mechanisms to recover from incomplete computation. Unlike these systems, we cannot count on a single central control and task distribution point to help discover and recover from incomplete computation. Instead we design a DAG execution structure that enforces synchronization points to catch substantial disruptions in computation or communication.

Mobile distributed computing: Projects in the mobile grid or mobile cloud area have some relationship to Serendipity; see Ahuja [1] for a survey. More recently, the Hyrax project envisions a somewhat similar capability to opportunistically use the resources of networked cellphones [25]. Our work is distinguished from these efforts in that we assume a much more disruptive network between devices.

9. CONCLUSION AND FUTURE WORK

In this paper we have presented the design and implementation of Serendipity, a general purpose distributed computing platform for DTNs. We have designed a simple yet powerful job structure suitable for distributed computing in the DTNs. A novel architecture is designed to help DTN nodes collaborate on distributed computing jobs. A set of task allocation and job scheduling algorithms are outlined for diverse DTN settings and distributed computing jobs. The extensive evaluation of Serendipity on Emulab demonstrates that it efficiently speeds up computationally complex applications.

As for the future, we will continue our investigation on distributed computing in DTNs in the following directions. First, we plan to deploy Serendipity on mobile devices (e.g., smart phones and tablets) for field tests. Second, we will implement more kinds of distributed applications on Serendipity. Based on the findings from this investigation, we will suggest improvements to further enhance the performance of Serendipity in DTNs.

10. REFERENCES

[1] S. Ahuja and J. Myers. A survey on wireless grid computing. *The Journal of Supercomputing*, 37:3–21, 2006. 10.1007/s11227-006-3845-z.

[2] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, November 2002.

[4] F. Bai and a. A. H. Narayanan Sadagopan. IMPORTANT: A framework to systematically analyze the impact of mobility on performance of routing protocols for adhoc networks. In *INFOCOM*, 2003.

[5] A. Balasubramanian, R. Mahajan, A. Venkataramani, B. N. Levine, and J. Zahorjan. Interactive wifi connectivity for moving vehicles. ACM SIGCOMM '08, pages 427–438, New York, NY, USA, 2008. ACM.

[6] A. Balasubramanian, Y. Zhou, W. B. Croft, B. N. Levine, and A. Venkataramani. Web search from a bus. ACM CHANTS '07, pages 59–66, New York, NY, USA, 2007.

[7] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[8] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *IEEE INFOCOM*, 2006.

[9] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: an approach to interplanetary internet. *Communications Magazine, IEEE*, 41(6):128 – 136, 2003.

[10] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. *Heterogeneous Computing Workshop*, 0:349, 2000.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[12] M. Demmer and K. Fall. DTLSR: delay tolerant routing for developing regions. NSDR '07, pages 5:1–5:6, New York, NY, USA, 2007. ACM.

[13] P. J. Denning. Hastily formed networks. *Commun. ACM*, 49:15–20, April 2006.

[14] K. Fall and S. Farrell. DTN: an architectural retrospective. *IEEE Journal on Selected Areas in Communications*, 26(5):828 –836, 2008.

[15] K. Fall, G. Iannaccone, J. Kannan, F. Silveira, and N. Taft. A disruption-tolerant architecture for secure and efficient disaster response communications. In *7th International ISCRAM Conference*, May 2010.

[16] S. Guo, M. H. Falaki, E. A. Oliver, S. Ur Rahman, A. Seth, M. A. Zaharia, and S. Keshav. Very low-cost internet access using kiosknet. *SIGCOMM Comput. Commun. Rev.*, 37:95–100, October 2007.

[17] K. M. Hanna, B. N. Levine, and R. Manmatha. Mobile Distributed Information Retrieval For Highly Partitioned Networks. In *IEEE ICNP*, pages 38–47, Nov 2003.

[18] P. Hui, J. Scott, J. Crowcroft, and C. Diot. Haggle: a networking architecture designed around mobile users. In *WONS*, 2006.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[20] W. Ivancic, L. Wood, P. Holliday, W. Eddy, D. Stewart, C. Jackson, and J. Northam. Experience with delay-tolerant networking from orbit. In *ASMS 2008*, pages 173 –178, 2008.

[21] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. *SIGCOMM Comput. Commun. Rev.*, 34:145–158, August 2004.

[22] W. jen Hsu, a. K. P. Thrasyvoulos Spyropoulos, and A. Helmy. Modeling time-variant user mobility in wireless mobile networks. In *INFOCOM*, 2007.

[23] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transaction on Acoustics, Speech and Signal Processing*, 1990.

[24] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104 –111, June 1988.

[25] E. Marinelli. Hyrax: Cloud computing on mobile devices using mapreduce. Master's thesis, Computer Science Dept., CMU, September 2009.

[26] P. Marshall. DARPA progress towards affordable, dense, and content focused tactical edge networks. In *IEEE MILCOM 2008*, pages 1 –7, 2008.

[27] S. Parikh and R. Durst. Disruption tolerant networking for marine corps CONDOR. In *IEEE MILCOM 2005.*, pages 325 –330 Vol. 1, 2005.

[28] A. S. Pentland, R. Fletcher, and A. Hasson. DakNet: Rethinking connectivity in developing nations. *Computer*, 37:78–83, January 2004.

[29] I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the levy-walk nature of human mobility. In *INFOCOM*, 2008.

[30] C. Rigano, K. Scott, J. Bush, R. Edell, S. Parikh, R. Wade, and B. Adamson. Mitigating naval network instabilities with disruption toler. In *IEEE MILCOM 2008*, pages 1 –7, 2008.

[31] A. K. Saha and D. B. Johnson. Modeling mobility for vehicular ad-hoc networks. In *Proceedings of the 1st ACM international workshop on Vehicular ad hoc networks*, VANET '04, 2004.

[32] K. Scott and S. Burleigh. Bundle protocol specification. *RFC 5050*, 2007.

[33] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav. Low-cost communication for rural internet kiosks using mechanical backhaul. In *ACM MobiCom '06*, pages 334–345, New York, NY, USA, 2006. ACM.

[34] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurr. Comput. : Pract. Exper.*, 17:323–356, February 2005.

[35] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment

for distributed systems and networks. In *USENIX OSDI*, 2002.

[36] Z. Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: overview and challenges. *Communications Surveys Tutorials, IEEE*, 8(1):24 –37, 2006.